Warped-MC: An Efficient Memory Controller Scheme for Massively Parallel Processors

Jong Hyun Jeong
Korea University
Seoul, South Korea
dida1245@korea.ac.kr

Myung Kuk Yoon
Ewha Womans University
Seoul, South Korea
myungkuk.yoon@ewha.ac.kr

Yunho Oh
Korea University
Seoul, South Korea
yunho_oh@korea.ac.kr

Gunjae Koo
Korea University
Seoul, South Korea
gunjaekoo@korea.ac.kr

ABSTRACT

The performance of GPU's external memories is becoming more critical since a modern GPU runs thousands of concurrent threads that demand a huge volume of data. In order to utilize resources in the memory hierarchy more efficiently, GPU employs a memory coalescing scheme to reduce the number of demand requests created from a group of threads (i.e. a warp). However, GPU's memory coalescing does not work well for applications that exhibit irregular memory access patterns, thus a single warp can generate multiple memory transactions. Since memory requests are serviced by different hierarchy levels and/or memory partitions, multiple outstanding requests from a single warp exhibit diverged fetch latency. Considering the execution time of a load warp is decided by the slowest memory transaction, the diverged memory latency within a warp is a critical performance factor for load warps.

In this paper, we propose a warp-aware memory controller scheme, called Warped-MC, to mitigate the memory latency divergence issues. Based on the in-depth analysis, we reveal the memory latency divergence within a warp is mainly caused by GPU memory controllers. While the conventional FR-FCFS memory controller can maximize the effective bandwidth of DRAM channels, the scheduling scheme of the conventional memory controller can exacerbate the memory latency divergence of a warp. Warped-MC employs a warp-aware scheduling scheme to alleviate the memory latency divergence, thus Warped-MC can tackle the long tail of the load warp execution time to improve the performance of memory-intensive applications. We implement Warped-MC on GPGPU-Sim configured with the modern GPU architecture, and our evaluation results exhibit Warped-MC can improve the performance of memory-intensive applications by 8.9% on average with a maximum of 45.8%.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**; *Processors and memory architectures.*

KEYWORDS

GPU Architecture, Memory System, Memory Controller

ICPP 2023, August 7–10, 2023, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0843-5/23/08…$15.00
https://doi.org/10.1145/3605573.3605645

ACM Reference Format:
Jong Hyun Jeong, Myung Kuk Yoon, Yunho Oh, and Gunjae Koo. 2023. Warped-MC: An Efficient Memory Controller Scheme for Massively Parallel Processors. In *52nd International Conference on Parallel Processing (ICPP 2023), August 7–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605645

1 INTRODUCTION

Graphics processing units (GPUs) exploit massive thread-level parallelism (TLP) to achieve high computation capability for parallel applications. GPUs have evolved to include more compute cores over generations to run more threads concurrently. Modern GPU systems demand high-performance memory subsystems since a modern GPU can run thousands of concurrent threads. In order to handle a huge number of demand requests created from many concurrent threads, GPU employs a memory coalescing scheme. Namely, a coalescing engine in a load/store (LDST) unit merges dozens of memory transactions created from a single warp (a group of 32 threads) into one or two wide memory transactions if demanded address spaces can be grouped into a single large block. However, modern applications such as machine learning applications and big data analytics exhibit irregular memory access patterns. For these applications the coalescing unit cannot merge memory transactions well, thus a single warp can create many memory transactions [5, 8, 21, 35]. Furthermore, GPU's on-chip memory size is small compared to the number of concurrent threads, thus many demand requests cannot be efficiently serviced by GPU's cache hierarchy. Hence, GPU kernels that exhibit irregular memory access patterns can create lots of off-chip memory accesses.

When a warp issues load instructions, the execution of the warp stalls until all the demand requests from the warp are serviced by GPU's memory hierarchy. Even though GPU can issue other available warps if a current warp is not available for execution, the long latency of off-chip accesses cannot be hidden by GPU's quick context switch among warps. Thus GPU pipelines frequently stall for long cycles if warps create many off-chip memory accesses [10, 34]. The latency of off-chip memory accesses can be elongated further due to severe congestion in memory systems. Off-chip accesses created from concurrent warps are aggregated in target memory partitions, then the memory controller associated with the target memory channel handles the aggregated pending transactions. Since off-chip accesses created from hundreds of concurrent warps are distributed to *several* memory partitions, resources in memory partitions can be fully occupied if warps create multiple off-chip accesses per warp. Moreover, GPU's memory controller first services the transactions that can minimize DRAM service time. Such memory controller schemes applied to many pending requests provoke long-tail in the latency of off-chip accesses. Note that the slowest

memory transactions determine the performance of load warps since the execution of a load warp stalls until all demanded data are fetched from the memory hierarchy. Hence, the execution time of load warps increases as the timing gap between the fastest and the slowest transactions within a single warp (called *latency divergence* in this work) is elongated. Consequently, memory latency divergence within a warp can be critical for the performance of memory-intensive applications [10, 21, 24, 30, 34].

In this paper, we propose an efficient memory controller scheme for GPU, called Warped-MC. We extensively analyze memory transaction latency in GPU memory hierarchy levels to reveal the sources that provoke latency divergence. Our analysis exhibits the memory latency divergence within a warp is mainly caused by the scheduling scheme of the conventional memory controllers which prioritize the pending requests that can access buffered rows in DRAM. In order to tackle the memory divergence issues and long-tail in off-chip memory accesses, Warped-MC prioritizes critical transactions by monitoring the number of pending requests per warp. Warped-MC can effectively improve the performance of memory-intensive applications that exhibit irregular memory access patterns by mitigating the memory latency divergence issues. We summarize the contributions of this work as follows.

- We disclose memory latency divergence issues are mainly caused by memory controllers. Especially we reveal the row-hit/miss scheduling and bank scheduling schemes of the conventional memory controller provoke latency divergence severely.
- Based on the in-depth analysis, we propose an efficient warp-aware memory controller scheme that can mitigate memory latency issues and long-tail in off-chip memory accesses.
- In order to evaluate our proposed memory controller scheme, we implement Warped-MC on the cycle-accurate GPGPU simulator [19]. Our evaluation exhibits Warped-MC can effectively improve the performance of memory-intensive applications by mitigating memory latency divergence.

The remainder of this paper is organized as follows. Section 2 describes conventional GPU architecture and memory controller schemes. We analyze the latency divergence issues in off-chip memory transactions in Section 3. We describe the scheduling algorithm and the hardware architecture of Warped-MC in Section 4. Evaluation details are exhibited in Section 5. Section 6 discusses related work. We conclude in Section 7.

2 BACKGROUND

2.1 GPU Architecture

As shown in Figure 1, a single GPU chip equips tens of streaming multiprocessors (SMs) and several memory partitions associated with external device memories. GPU employs a single-instruction multiple-thread (SIMT) execution model to run multiple concurrent threads generated from the same kernel. GPU groups multiple threads (32 threads usually) to form a warp, and a single SM can run dozens of concurrent warps. In order to hide the latency of execution units, GPU employs fine-grained multithreading among concurrent warps. Namely, an SM can issue any available warps

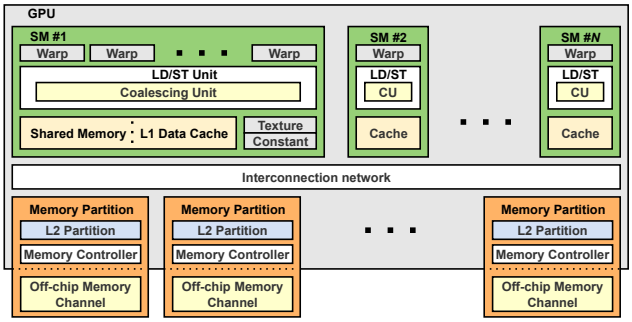


Figure 1: GPU architecture

if a current warp cannot be executed. However, off-chip memory accesses usually take hundreds of cycles, thus such long latency cannot be hidden by quick context switches among dozens of warps [22].

GPU's memory hierarchy is designed to efficiently handle a large number of memory transactions generated from many concurrent threads. A load/store (LDST) unit in an SM employs memory coalescing to reduce the number of memory transactions created from a single warp. Namely, the coalescing unit merges multiple memory transactions created from a warp into a single request if the addresses of the transactions can be allocated in a large memory block. Thus, a single warp can create one or two memory requests with memory coalescing if the warp accesses regularly allocated data. However, the memory coalescing does not work well if the treads within a warp exhibit irregular access patterns. In this case, a single warp creates many memory transactions to provoke severe congestion in the memory hierarchy. An SM equips a private L1 data cache shared by the concurrent threads in the SM. Modern GPU architectures employ a sector cache structure and a streaming cache structure for the L1 data cache to handle multiple irregular outstanding requests more efficiently [9, 18, 31]. Multiple memory partitions and tens of SMs are connected via an interconnection network as shown in Figure 1. Each memory partition includes one or more L2 cache blocks and memory channels to external memories. Note that L2 caches can be shared by multiple SMs via the interconnection network. In order to exploit memory-level parallelism (MLP), address space is interleaved across multiple memory partitions.

As a modern GPU equips more SMs to increase the number of concurrent threads, GPU's performance can be significantly restricted by the performance of the GPU memory hierarchy [11, 14]. Especially, memory-intensive kernels can create a large amount of off-chip accesses since GPU's cache hierarchy is not efficiently utilized due to cache pollution and severe congestion by excessive memory transactions. Note that off-chip accesses take longer as data traffic is heavy on memory channels. In this case, SM pipelines stall more frequently since the execution of load warps is blocked until the demand requests are serviced by external device memories.

2.2 GPU Memory Controller

GPU deploys GDDR [32] or high-bandwidth memory (HBM) [16] as a main device memory in order to provide high bandwidth for heavy data transactions demanded by thousands of concurrent threads.

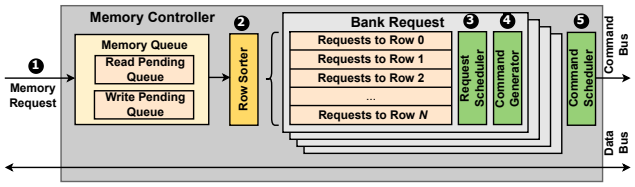


Figure 2: GPU memory controller scheme

Each main memory chip includes multiple banks that can operate independently to exploit bank-level parallelism (BLP). GPU usually employs an XOR-based bank address indexing policy to distribute memory requests from a single warp to multiple banks [13]. As shown in Figure 1 a single GPU chip is connected with multiple external DRAM chips via multiple memory channels associated with the corresponding L2 partitions. A memory controller manages the data transactions between the L2 partitions and the associated DRAM chips. Hence, GPU can provide high data bandwidth to the external DRAM by exploiting memory-level parallelism via multi-bank and multi-channel topology.

GPU's memory controller associated with each memory channel is designed to provide high bandwidth to the connected DRAM chips. Figure 2 depicts the first-ready, first-come-first-service (FR-FCFS) [29] memory scheduler scheme employed to GPUs. The FR-FCFS scheduling can provide higher bandwidth compared to a simple first-come-first-service (FCFS) scheme since FR-FCFS schedules the requests that will access the *same row* of a target bank with higher priority. Note that a memory controller can save *activate* and *precharge* commands since the requested data can be provided from a row buffer (i.e. sense amplifiers) quickly.

An FR-FCFS memory controller works as follows. When the memory controller receives a memory request from an L2 partition, the request is enqueued in the one of pending queues (❶). Then, the controller identifies the bank ID and the row address of the request in the pending queue to sort the request by bank IDs (❷). The controller schedules the request with higher priority if the row address of the request is equal to the address of the open row (i.e. row data in a row buffer). If the controller cannot find any requests that can access the data in the row buffer, the controller schedules the oldest request to the target bank (❸). This is a row conflict case, and the controller needs to issue additional DRAM commands to close the current row and fetch another row [26]. When a request is scheduled, the controller creates DRAM commands based on required operations (❹), then the controller issues the commands to a target bank (❺). Normally the controller selects a target bank from an available bank pool in a round-robin fashion in order to guarantee fairness among multiple banks. To summarize, an FR-FCFS memory controller tends to issue the requests to the open row with higher priority regardless of the waiting time and the urgency of the pending requests.

3 MOTIVATION

3.1 Latency Divergence in Memory Requests

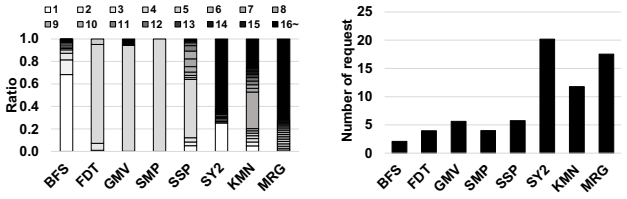
Since GPU runs hundreds of warps concurrently, the memory transactions created by many concurrent wraps are jumbled in GPU's memory hierarchy. If requested data is not found in the target L2 partition, the missed transactions need to be serviced by the memory controllers associated with the L2 partition. As described in Section 2.2, GPU's memory controllers adjust the issuance order of pending requests to maximize the throughput of external memories. Namely, the memory controllers do not consider any information regarding warp executions for scheduling memory requests. Note that the latency of a load warp is decided by the slowest memory transaction. In the worst-case scenario, a load warp cannot complete for an extremely long cycle even if only one memory request is not serviced due to the *first-ready* scheme of the memory controller. In this paper, we define memory latency divergence as a gap between the shortest and the longest turnaround times of the off-chip memory requests generated from a single warp. Obviously, a load warp can be committed after the last memory request is serviced, thus the memory latency divergence of a load warp is critical for the performance of warp execution [20, 21, 30].

We analyze the memory latency divergence observed in GPU applications using 14 benchmarks selected from the popular GPGPU benchmark suites [7, 12, 23, 33]. Based on the amount of off-chip memory accesses created from a single load warp, we classify the GPU applications as low off-chip access (LOA), medium off-chip access (MOA), and high off-chip access (HOA) applications. For LOA applications, less than one off-chip access is generated from a load warp on average. We can observe more than 8 off-chip accesses per warp on average for HOA applications, thus such applications provoke extremely heavy traffic on the memory hierarchy that GPU performance is restricted by the limited bandwidth of memory channels [10]. MOA applications create 2–8 off-chip accesses per load warp on average.

Table 1: Benchmarks

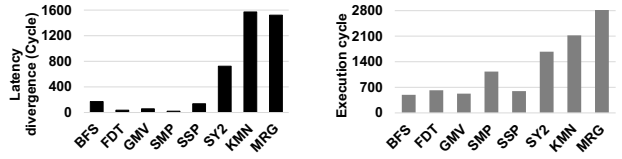
Abbr.	Description	Type
2MM	2 Matrix Multiplications [12]	LOA
3MM	3 Matrix Multiplications [12]	LOA
GAS	Gaussian Elimination [7]	LOA
DIT	Multiresolution Analysis kernel [12]	LOA
NSP	Survey Propagation [23]	LOA
MRQ	Magnetic Resonance Imaging-Q [33]	LOA
BFS	Breadth-First Search [7]	MOA
FDT	2D Finite Difference Time Domain Kernel [12]	MOA
GMV	Vector Multiplication and Matrix Addition [12]	MOA
SMP	Stream Priorities [27]	MOA
SSP	Single-Source Shortest Path [23]	MOA
SY2	Symmetric Rank-2K Operation [12]	HOA
KMN	K-means [7]	HOA
MRG	Magnetic Resonance Imaging-Gridding [33]	HOA

We investigate the characteristics of off-chip accesses per warp for MOA and HOA applications as shown in Figure 3. Note that memory latency divergence is not critical for LOA applications since load warps are well-coalesced and less than one request per warp is serviced by DRAM. We can observe several warps create more than 10 off-chip accesses for BFS and GMV. For HOA applications, a large fraction of warps creates many off-chip accesses since threads in a warp exhibit irregular access patterns and many memory transactions from uncoalesced warps miss in the cache hierarchy.



(a) Distribution of off-chip accesses per warp (b) Average number of off-chip accesses per warp

Figure 3: Off-chip access characteristics



(a) Average memory latency divergence (b) Average load warp execution time

Figure 4: Latency divergence and warp execution time

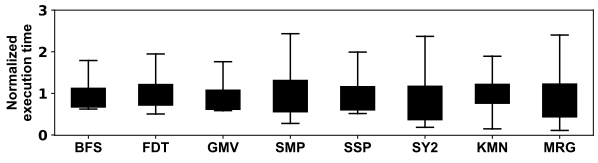


Figure 5: Distribution of load warp execution cycles

Figure 4 exhibits the average memory latency divergence and the execution time of the load warps that generate multiple off-chip accesses. We observe HOA applications exhibit extremely high memory latency divergence. For KMN the average latency divergence is 1570 cycles, which means a load warp needs to wait for additional 1570 cycles on average since the fastest request is serviced. Our observation reveals the GPU applications that create more off-chip accesses exhibit severe memory latency divergence. As shown in Figure 4b the load warps of such applications also exhibit longer execution time.

The box plot in Figure 5 exhibits the distribution of load warp execution cycles for MOA and HOA applications. The execution cycle of each load warp is normalized to the average execution cycle of entire load warps for each application. The upper and lower horizontal lines of a black-colored box represent the lower quartile (25th percentile) and the upper quartile (75th percentile) of warp execution time. As shown in the figure we observe a wide range of distribution in the execution cycles of load warps. Especially we can observe severe long-tail [1] (upper 25% range) of load warp execution time. If a load warp exhibits extremely long execution cycles, a warp scheduler cannot find any available warps thus the pipelines in an SM can stall for many cycles. We can say such long-tail of a load warp execution time is mainly caused by the latency divergence in off-chip memory accesses. We now investigate the sources of memory latency divergence in the next section.

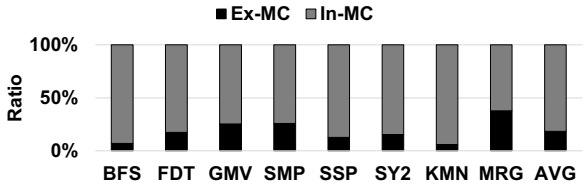


Figure 6: Fraction of memory latency divergence by Ex-MC and In-MC

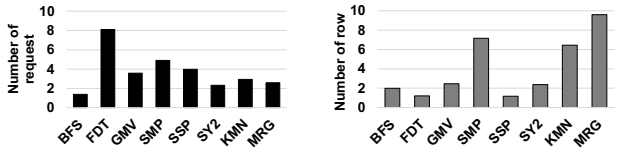
3.2 Sources of Memory Latency Divergence

As explained in Section 2.1, memory requests from a warp first access the private L1 cache in an SM and then missed requests are transferred to the shared L2 partitions. The transactions missed in the L2 cache are delivered to the associated memory controller to access external DRAM. Since memory requests pass through several hardware components in multiple memory hierarchy levels, these hardware resources can lead to latency divergence observed in multiple memory requests from a warp. In this section, we analyze the sources of memory latency divergence in GPU memory hierarchy. We classify the sources of the memory latency divergence into two parts: external parts of a memory controller (*Ex-MC*) and internal parts of a memory controller (*In-MC*).

Ex-MC: Multiple memory requests from a warp access L1 cache, interconnection network, and L2 cache partitions before the requests are serviced by memory controllers. The hardware components within this path can cause latency divergence among multiple requests. For instance, outstanding requests registered in L1 miss status holding registers (MSHRs) are sent to the injection queue of the interconnection network sequentially, however, the later requests can be injected after many cycles due to the queuing delays and the scheduling policy of the interconnection network. Furthermore, an additional cycle gap can be exhibited between earlier and later transactions since the memory request from an SM can be mixed with many requests from other SMs in a destination L2 partition [36]. Latency divergence can be also caused by different congestion levels (i.e. different counts of waiting transactions) across multiple L2 partitions.

In-MC: Memory latency divergence can be provoked by a memory controller since GPU memory controllers usually employ out-of-order schemes to increase effective bandwidth to external DRAM. As described in Section 3.3 the FR-FCFS DRAM controller schedules the requests that access the open-row with the highest priority, thus the pending requests that need to change the buffered row may not be serviced for many cycles. Such requests can exhibit a longer turnaround time. Moreover, DRAM timing constraints and bank scheduling can increase the timing gap between consecutive memory requests [25]. In the following section, we will describe the detailed scheduling schemes that can aggravate the memory latency divergence in the memory controller.

Figure 6 exhibits the fraction of memory latency divergence provoked by Ex-MC and In-MC. For memory transactions created from a warp that accesses DRAM, we measure the latency of hardware components in GPU memory hierarchy to compare the latency between the fastest and the slowest memory requests. For instance, In-MC represents the latency difference between the fastest and



(a) Average number of consecutive requests in an open row **(b) Average number of replaceable rows**

Figure 7: Average number of replaceable rows and consecutive requests in an open row

the slowest requests in a memory controller. Our evaluation reveals that latency differences in memory controllers occupy the most part of the memory latency divergence for MOA and HOA applications. On average 82.6% of memory latency divergence is provoked by the memory controllers. Our analysis results exhibit the out-of-order scheduling of the FR-FCFS controllers is the most critical factor that results in the memory latency divergence.

3.3 Analysis of Memory Controller Scheduling

In this section, we analyze the detailed scheduling mechanisms of the FR-FCFS memory controller, which schedules the pending requests that will hit the open-row first and then apply conventional FCFS scheduling. Note that latency divergence is caused by the timing gap between consecutive requests. We cannot avoid the timing differences by DRAM timing constraints since such timing properties are native characteristics of DRAM circuits. On the other hand, we can adjust the conventional scheduling algorithms to alleviate memory latency divergence. We investigate three scheduling parts that can be modified for mitigating divergent turnaround times of memory requests within a warp.

Request scheduling for row-hit cases: If the FR-FCFS controller finds the pending requests that hit the open-row of the target bank, the controller schedules these requests first. In order to access column data in the row buffer of the target bank, the memory controller issues a column address strobe (CAS) command to DRAM. Since a CAS command consumes 5.71 ns for GDDR6 [32], a request needs to wait for long cycles in the pending queues if there are many earlier requests that will hit the buffered row. Note that the FR-FCFS controller employs an FCFS scheme for the requests that will access open-row. Figure 7a exhibits the average number of pending requests that can hit a buffered row when a row buffer size is 1 KB. For MOA and HOA applications the memory controller can find several pending requests that will access the open-row of a target DRAM bank. The maximum number of *ready* requests is over 60 for many applications when we configure large pending queues. Our results represent we can set higher priority for more urgent requests if the controller finds multiple pending requests that will hit the open-row.

Row scheduling for row-miss cases: The FR-FCFS controller schedules *first-arrived* requests for generating DRAM commands if the controller cannot find any requests that will hit an open-row. In this case, the controller needs to issue additional DRAM commands such as a *precharge* command for closing the open-row and an *activate* command for transferring the target row to the row buffer. Moreover, modern DRAM devices require additional timing constraints such as row-to-row delay (RRD) and read-to-precharge (RTP) delay [32]. Thus DRAM controllers consume more cycles for row-miss cases compared to row-hit cases. Hence, late requests can be delayed longer if the controller handles row-miss cases. The conventional controller employs an FCFS scheme for selecting a request that replaces an open-row. However, this scheme can aggravate memory latency divergence since *lately-arrived* requests wait longer while former requests are scheduled for a row-miss case. Figure 7b exhibits the average number of requests that can be selected for row-miss cases. Our analysis reveals there are multiple pending requests that can replace an open-row of a target bank, thus later requests wait for many cycles due to multiple row replacements.

Bank scheduling: As described in Section 2.2, a conventional memory controller selects a target bank from an available bank pool in a round-robin fashion. Even though the round-robin scheme can guarantee fairness across multiple banks, this scheme cannot expedite the bank that includes urgent requests since the conventional controller does not consider the architectural features of processor cores. For instance, in a round-robin fashion, the memory controller set the lowest priority for the bank that performed DRAM commands most recently, thus the memory requests that target this bank can be serviced after all other banks receive DRAM commands even if the request needs to be serviced quickly.

To summarize, the conventional FR-FCFS memory scheduler is designed for maximizing the effective bandwidth of DRAM channels since the throughput of memory systems is critical for the performance of GPU. However, the conventional controller does not consider the architectural features of GPU, thus the scheduling schemes of the current DRAM controllers can exacerbate the memory latency divergence within a warp to slow down the execution of load warps. Based on an in-depth analysis of the conventional memory controller schemes on GPU, we disclose several scheduling parts that can be modified for mitigating latency divergence caused by the memory controller.

4 WARPED-MC

Albeit memory latency divergence within a warp can be one of the critical factors that increase the latency of load warps, a conventional memory controller scheme does not consider the SIMT execution model of GPUs. Namely, the GPU memory controller cannot expedite the slowest memory requests within a warp even though the slowest transaction decides the execution latency of a load warp. In order to tackle the latency divergence issues caused by the conventional memory controller, we propose a warp-aware memory controller scheme, called Warped-MC. In order to alleviate the memory latency divergence of a warp, Warped-MC adjusts the priorities of slower requests among multiple memory requests generated from a single warp. Warped-MC does not sacrifice the bandwidth of memory channels since DRAM bandwidth is one of the critical performance factors of modern GPU architectures. Warped-MC receives minimum warp information from SMs to fine-tune the priority of pending memory requests. In this section we first describe the scheduling algorithm of Warped-MC, then we explain the hardware architecture of Warped-MC.

4.1 Warped-MC Scheduling Algorithm

As discussed in Section 3.3, the memory latency divergence of a warp can result from the scheduling policies of the conventional memory controller. The *first-ready* scheme can lead to memory latency divergence since the memory controller first schedules the requests that will hit an open-row without considering GPU's warp structures. However, the *first-ready* scheme of the FR-FCFS controller is an essential feature that can improve the effective bandwidth of memory channels, thus Warped-MC tackles the *FCFS* scheduling schemes of a memory controller to mitigate the memory latency divergence of a warp. Algorithm 1 represents the request scheduling for row-hit cases, row scheduling for row-miss cases, and bank scheduling schemes of Warped-MC.

Algorithm 1 Warped-MC scheduling

```
1:  warp ID of warp A ← wid
2:  bank ID of request ← bid
3:  row ID of request ← rid
4:  procedure REQUEST PRIORITY
5:      if schedule request then
6:          warp_entry[wid].n_pending−−
7:          if warp_entry[wid].n_pending == 1 then
8:              last request ← priority H
9:          if warp_entry[wid].n_pending ≥ 2 then
10:             pending requests of warp A ← priority M
11: procedure ROW PRIORITY
12:     if schedule request then
13:         if warp_entry[wid].n_pending == 1 then
14:             row_scoreboard[bid][rid]++
15: procedure BANK PRIORITY
16:     if bank is idle then
17:         bank_priority ← scheduled_request.priority
```

Request priority: Even though a row-hit case is the best situation that can reduce the number of DRAM commands, consecutive CAS commands consume several DRAM cycles thus later requests need to wait for dozens of cycles. Note that a memory controller has to expedite the slowest request from a warp in order to alleviate memory latency divergence. Hence, if Warped-MC finds multiple pending requests that will hit an open-row and one of the requests is the slowest request of a warp, Warped-MC schedules the slowest request with higher priority. If Warped-MC finds multiple pending requests left in the warp after scheduling a request and those requests hit the open-row, Warped-MC schedules the pending requests with second priority. Since the pending requests have to be scheduled consecutively to finish the warp quickly. The *request priority* procedure in Algorithm 1 represents the Warped-MC scheduling scheme for this case. In order to detect the slowest request of a warp, Warped-MC monitors the number of pending requests identified by warp IDs. If a request in the pending queues is the last pending request of a warp (i.e. the number of pending requests of a warp is one), Warped-MC increases the priority of the slowest request. If a request of the warp that has multiple pending requests in the pending queue hits the open-row, Warped-MC elevates the priority of the request to the second level. Thus, Warped-MC can schedule the urgent request of a warp with higher priority when *ready* pending requests are scheduled.

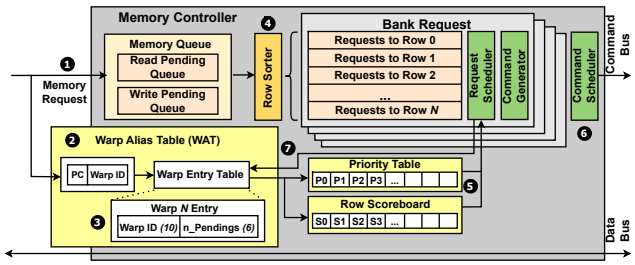


Figure 8: Hardware architecture of Warped-MC

Row priority: Whereas the conventional memory controller employs a simple *FCFS* scheme, Warped-MC schedules the row that includes the slowest request of a warp with higher priority in case of row-misses. For this purpose, Warped-MC groups pending requests by row IDs (i.e. a pending row) to record scores for pending rows based on the urgency of requests. Warped-MC increases the score of a pending row if the row includes the slowest requests of a warp as shown in the *row priority* procedure of Algorithm 1. When row replacement is needed in case of a row-miss, Warped-MC selects the pending row that exhibits the highest score as a new open-row. Once a buffered row is replaced, Warped-MC schedules the requests in the selected pending row based on the *request priority* procedure. Then the score of the selected row becomes zero.

Bank priority: Warped-MC set higher priority for the banks that will schedule more urgent requests rather than a conventional round-robin fashion. Note that the request scheduler sets the higher priority for the slowest request that will hit the row buffer of the target bank. The command scheduler of Warped-MC first selects an available bank that can receive DRAM commands from the higher-priority bank pool in a round-robin fashion, then the command scheduler scans the lower-priority bank pool to pick an available bank. This two-level scheduling mechanism of Warped-MC is helpful for mitigating the memory latency divergence in a warp since the command scheduler checks a smaller number of high-priority banks whereas the conventional memory controller scans entire banks (16 banks for GDDR6) in a round-robin fashion [32].

4.2 Warped-MC Architecture

Figure 8 depicts the hardware architecture of Warped-MC. Warped-MC also relies on the *first-ready* scheme like the conventional FR-FCFS controller in order to maximize the effective bandwidth of DRAM channels (see the *request priority* part in Section 4.1). In order to implement the warp-aware scheduling schemes, Warped-MC includes the additional hardware components as follows. A warp alias table (WAT) creates a global warp ID for memory controllers by combining a hashed PC of a load, a warp ID in an SM, and an SM ID. Note that Warped-MC identifies the load warps that create multiple off-chip memory accesses to apply warp-level control schemes. A warp entry table (WET) stores the status of load warps managed by Warped-MC. The status information such as the number of pending requests (*n_pendings*) is indexed by the global warp ID created by WAT, thus Warped-MC can access the status information using the global warp ID. WAT and WET are global

hardware components shared by memory controllers in L2 memory partitions. Each memory controller of Warped-MC includes private components such as a row scoreboard (RS) and a priority table (PT). Warped-MC stores the scores of pending rows per table in RS. Warped-MC makes use of PT for identifying the priority of each bank, thus each entry of PT is associated with the corresponding bank.

We represent the data and control flows of Warped-MC using arrows and circled numbers. When the memory controller of Warped-MC receives a memory request from an associated L2 cache, the request is enqueued in the one of pending queues (❶). Simultaneously, WAT creates a global warp ID to register the status information of the warp in WET (❷). If a memory request is firstly registered to WET using a new global warp ID, the request status information of the warp is initialized with *n_pendings=1* (❸). If WET already has an entry of a warp, only the *n_pendings* field is incremented. When a new memory request is enqueued in a pending queue, the row ID and the bank ID of the request are decoded and then such information is given to the per-bank scheduler of Warped-MC (❹). As described in the previous section, for row-hit cases Warped-MC first schedules the requests that will hit an open-row based on the request scheduling scheme (see *request priority* of Algorithm 1).

Warped-MC manages the private tables (RS and PT) in each memory controller using the warp status information in WET. If the *n_pendings* field of a warp is one, which means the received request is the last (i.e. slowest) request of the corresponding warp or the warp creates only one off-chip request. As explained in the previous section, Warped-MC expedites such requests to alleviate the memory latency divergence of a warp. When a pending request is decoded by the row sorter, Warped-MC references WET using a global warp ID and then updates RS indexed by the decoded row ID. When a memory controller replaces an open-row of a target bank (i.e. a row-miss case), Warped-MC picks the request in the pending row that exhibits the highest row score as the next request to be serviced by the target bank (❺). Warped-MC selects the target bank of DRAM commands based on the two-level bank scheduling scheme as explained in the previous section. In order to classify high-priority and low-priority banks, Warped-MC records the priority data of the requests per bank in each entry of PT. Warped-MC picks a target bank based on the two-level bank scheduling scheme and then injects generated DRAM commands to the associated DRAM channel (❻). Once a request is serviced from DRAM, the request is evicted from the pending queue and WET entry of the corresponding warp is updated (❼)

5 EVALUATION

5.1 Experiment Setup

We implement Warped-MC on the cycle-accurate GPU simulator, GPGPU-Sim v4.2 [4, 19]. We estimate the power consumption of GPU using AccelWattch [17]. We configure the GPGPU-Sim using the configuration parameters listed in Table 2. The baseline GPU architecture used for the evaluations in this work is similar to the configurations of NVIDIA RTX 2060 Super [28]. The timing parameters of GDDR6 are scaled based on the memory clock rates listed in Table 2 [32]. We compare the performance of Warped-MC with the baseline memory controller scheme that employs FR-FCFS scheduling. We evaluate Warped-MC using the GPU benchmarks listed in Table 1. All target applications are simulated until the end of execution or until the number of committed instructions reaches 1 billion.

Table 2: GPU configurations

Parameter	Configuration
Core	32 SMs, 64 CUDA cores / SM @1905MHz
Warp	32 Warps / SM
Warp scheduler	LRR, 4 schedulers / SM
CTA	32 CTAs / SM
Register file	256KB / SM
L1 data cache	64KB / SM, 128B line, 4 sector / line, 512 way, LRU, 256 MSHRs
L2 cache	128KB × 32 partitions, 128B line 16 way, LRU, 192 MSHRs / partition
DRAM	384bit bus @3500Hz, 16 channels, 1 controller / channel 16 banks / chip, 1KB row size
GDDR timing	tRP=20, tRC=62, tRAS=50, tRCD=20, tRRD=10,tWR=20, tCL=20, tCCD=4

5.2 Performance

Figure 9 exhibits the performance of Warped-MC. We measure the instruction per cycle (IPC) of each application by applying different memory controller schemes. IPCs by other memory controllers are normalized to the IPC of the baseline (i.e. FR-FCFS controllers). FCFS controllers guarantee fairness among pending requests, however, the FCFS controllers cannot achieve high bandwidth on memory channels since buffered rows are replaced more frequently in DRAM banks. Div-First implements a scheduling scheme that prioritizes urgent transactions first without considering row-hit cases. Our evaluation exhibits Warped-MC improves the performance of all types of applications by 5.1% on average. For MOA and HOA applications, Warped-MC uplifts the performance by 8.9% on average with a maximum of 45.8% (KMN). FCFS controllers degrade GPU performance by 28.3% for all types of applications and 43.2% for MOA and HOA applications compared to FR-FCFS controllers. These evaluation results present that effective bandwidth in memory channels is a critical performance factor for modern GPU architectures. As such, Div-First exhibits 6.0% lower performance on average compared to the baseline since Div-First focuses on mitigating latency divergence without considering row-hit cases. On the other hand, Warped-MC can improve the performance of modern GPU architectures by tackling latency divergence issues without downgrading the effective bandwidth of memory channels.

We further analyze how Warped-MC adjusts the scheduling of pending requests compared to the baseline controllers. Figure 10 exhibits the breakdown of the scheduling operations adjusted by Warped-MC. As described in Section 4.1, Warped-MC adjusts the priorities of pending requests for row-hit cases, row scheduling for row-miss cases, and bank scheduling to alleviate memory latency divergence. For MOA and HOA applications, Warped-MC changes the bank scheduling frequently by setting higher priorities for the banks that include urgent requests. For HOA applications, Warped-MC adjusts the row scheduling more frequently since these

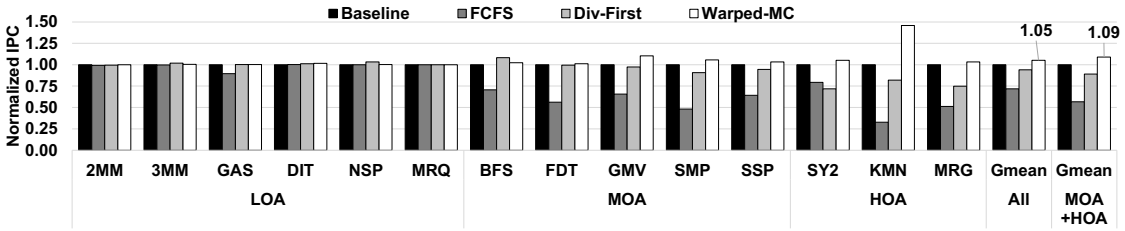


Figure 9: Performance normalized to the baseline

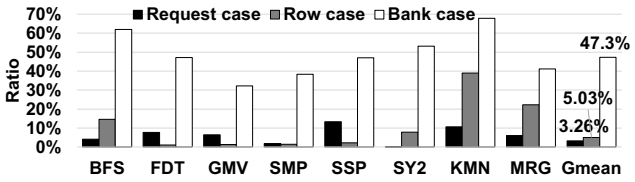


Figure 10: Ratio of rescheduling operations by Warped-MC

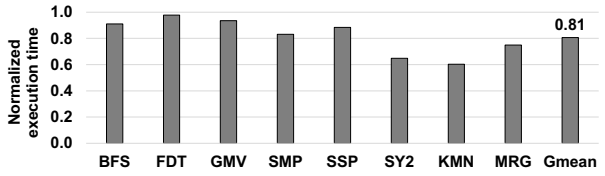


Figure 11: Execution time of load warps normalized to the baseline

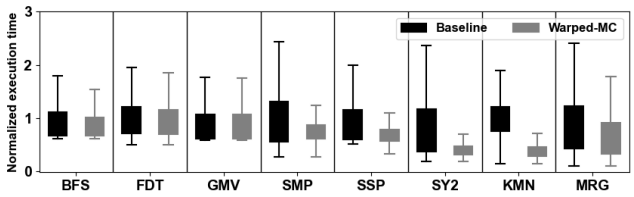


Figure 12: Distribution of load warp execution cycles

applications generate many off-chip transactions that access multiple rows in the same bank. The request rescheduling by Warpe-MC is relatively small since Warped-MC finds a small number of requests that will hit an open-row.

5.3 Warp Execution Time

Warped-MC curtails the execution cycles of load warps by alleviating memory latency divergence. Figure 11 shows the average execution time of load warps by Warped-MC for MOA and HOA applications. The measured warp execution cycles are normalized to the average execution cycles of load warps on the baseline system. Since Warped-MC expedites the services for urgent off-chip accesses, Warped-MC can effectively reduce the turnaround time of the slowest memory transaction within a load warp. Hence, the overall execution cycles of load warps can be reduced by Warped-MC. Our evaluation results exhibit that Warped-MC decreases the execution cycles of load warps by 19.3% on average with a maximum of 39.7% for MOA and HOA applications.

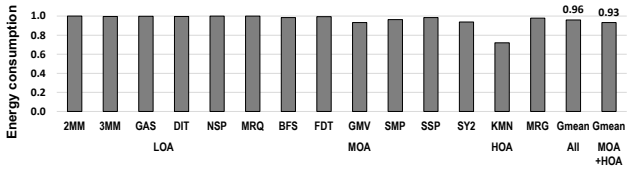


Figure 13: Energy consumption normalized to the baseline

Moreover, Warped-MC improves the overall performance of load warps by reducing the long-tail of load warp executions significantly. Figure 12 exhibits the distribution of load warp execution cycles for MOA and HOA applications. For each application, the execution cycles of load warps are normalized to the average execution cycle of load warps by the baseline FR-FCFS controller. The upper and lower horizontal lines of a box represent the lower quartile and the upper quartile of the distribution of execution cycles. The black-colored box plot and the grey-colored box plot represent the distributions by the FR-FCFS controller and Warped-MC respectively. Our evaluation reveals that Warped-MC reduces the long-tail of load warp executions significantly. Especially the upper 25% range of the distributions by Warped-MC is in the range of 25% to 75% of distributions (i.e. inside of a box) by the baseline for SMP, SSP, SY2, and KMN. Consequently, Warped-MC significantly improves the performance of load warps by tackling the memory latency divergence provoked by memory controllers.

5.4 Energy Efficiency

Figure 13 shows the energy consumption by Warped-MC. The energy consumption of each application is normalized to the energy consumption by the baseline machine that employs FR-FCFS memory controllers. Our evaluation results exhibit Warped-MC reduces the energy consumption of GPU by 4.0% for all types of applications and 6.7% for MOA and HOA applications. The reduction of energy consumption by Warped-MC benefits from the reduced execution cycles. Since Warped-MC minimizes the row exchanges by scheduling requests to an open-row first, Warped-MC does not increase the power consumption in DRAM. Our evaluation results reveal Warped-MC can improve energy efficiency in modern GPU architectures.

5.5 Hardware Overhead

In order to implement a warp-aware memory controller scheme, Warped-MC includes additional hardware components such as WAT, RS, and PT as shown in Figure 8. Table 3 lists the size of the additional hardware components for implementing Warped-MC on the baseline GPU configurations (similar to NVIDIA RTX 2060 Super) exhibited in Table 2.

Table 3: Hardware overhead by Warped-MC

Component	Entry size	No. of entries
WET	6 bits	1024 entries
RS	256 bits	16 entries
PT	64 bits	16 entries

Warped-MC exploits the information of warp-level pending requests to set different priority levels to the pending requests creased from the corresponding warp. WAT generates a warp ID that is used for accessing WET by combining an SM ID and a warp ID per SM. Since the baseline GPU equips 32 SMs and each SM runs up to 32 concurrent warps, WAT creates a 10-bit warp ID. Then, the number of pending requests is logged in WET entries indexed by the warp ID. A single warp can create a maximum of 32 off-chip accesses if each thread within the warp creates a separate outstanding request (i.e. fully-uncoalesced). Hence, the size of a single WET entry is 6 bits. Each entry of RS accommodates scores for rows in the corresponding bank. We set the size of a per-row score as 4 bits. Thus, the size of a single RS entry is 256 bits since a single bank includes 64 rows with the baseline GPU configurations [32]. Note that the number of RS entries is equivalent to the number of banks per memory channel. The size of a single PT entry is determined by the maximum number of consecutive requests for an open-row. If a single row size is 1 KB, the size of a PT entry is 64 bits. Each entry of PT is associated with the corresponding bank. The size of the additional hardware components per memory controller is listed in Table 3. For the baseline configurations, Warped-MC requires 1,408B of additional storage space per memory controller.

6 RELATED WORKS

As GPU relies on massive TLP, the limited capacity and performance of the GPU memory hierarchy is one of the critical performance hurdles of GPU. As such, researchers have presented research outputs for improving the performance and efficiency of GPU memory systems.

6.1 Memory Controller

Since the GPU performance can be significantly restricted by the effective bandwidth of memory channels, researchers have presented efficient memory controller schemes that can handle many off-chip requests efficiently. Jog et al. presented a memory controller scheduling scheme that adjusts the priorities of off-chip accesses based on the latency tolerance of SMs [15]. Namely, the proposed memory controller set lower priorities if the pending requests are originated from the latency-tolerant SMs. On the other hand, our approach focuses on the latency divergence and the long-tail within a warp, thus Warped-MC prioritizes the urgent transactions based on the remaining pending requests per warp. Chatterjee et al. proposed a memory controller that gathers warp-based information globally to issue DRAM commands consecutively for the requests from the same warp [6]. This approach may alleviate the latency divergence issue, however DRAM commands can be issued inefficiently since the *first-ready* scheduling policy is underrated. Warped-MC can operate more efficiently because it tackles memory latency divergence while maintaining the high effective bandwidth of conventional controllers. Mu et al. represented prioritizing memory requests based on fixed DRAM turnaround time since the average memory latency varies by applications [25]. Warped-MC can be effective for any types of applications since Warped-MC monitors the information of pending requests from warps in runtime to prioritize urgent requests for mitigating memory latency divergence.

6.2 Cache Management

There are studies that are priority-based cache management schemes for mitigating latency divergence. Arunkumar et al. [2] proposed a cache bypassing mechanism based on reuse distance and cache management policy with adaptive cache granularity. However, in modern GPUs with architectural changes such as sectored cache, the tendency of reuse distance may be not dissimilar from before. Our work targets state-of-the-art GPUs and designs effective solutions for latency distribution based on in-depth analysis. Ausavarungnirun et al. [3] proposed a cache management mechanism to prevent to evict data of warp with a high cache hit rate (hit warp) and a memory scheduler to prioritize the pending requests of hit warps. In applications with irregular memory access patterns, the proportion of hit warps is low, making it difficult for this mechanism to apply. Because Warped-MC algorithm is based on the number of pending requests from the warp, it can be applied to irregular applications as well as regular applications. However, our work can be extended to combine with appropriate cache management policies.

6.3 Warp Scheduling

Meng et al. [24] proposed a dynamic warp subdivision (DWS), which allows a single warp to occupy more than one slot in the warp scheduler. DWS improves memory level parallelism and latency hiding by allowing hit threads in a warp to run in advance. Poise [11] by Dublish et al. is the mechanism that derives optimal warp scheduling with machine learning from profiled kernels to balance TLP and memory system performance. These approaches improve MLP by limiting TLP, making it difficult to use the GPU's computing resources fully. Warped-MC is a solution that does not sacrifice TLP but also increases active threads by reducing warps that are stalled due to long latency.

7 CONCLUSION

As a modern GPU accommodates thousands of concurrent threads, GPU performance can be significantly restricted by the limited resources in memory channels and external device memories. Since thread executions within a warp are explicitly synchronized, the performance of load warps is determined by memory transactions that exhibit the longest turnaround cycles in a warp. We observe services for such critical memory requests are even delayed due to the scheduling schemes of the conventional memory controller as well as queuing delays in the GPU memory hierarchy. For memory-intensive kernels that create excessive off-chip accesses, memory latency divergence within a warp downgrades the performance of load warps significantly. In this paper, we propose a warp-aware memory controller scheme, called Warped-MC, to tackle the memory latency divergence provoked by off-chip memory accesses. Based on the number of pending requests from a warp, Warped-MC set higher priorities for urgent memory transactions to make critical requests serviced from DRAM quickly. Warped-MC can

mitigate memory latency divergence effectively without sacrificing the throughput of memory channels. Our evaluation reveals Warped-MC uplifts the performance of memory-intensive applications by 8.9% on average by improving the distributions of load warp execution cycles.

ACKNOWLEDGMENTS

This work was supported in part by the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) of the Government of South Korea (NRF-2021R1C1C1012172, NRF-2022R1C1C1011021, and No. 2021R1G1A1092196) and in part by MSIT under the ICT Creative Consilience Program (IITP-2023-2020-0-01819) supervised by the Institute for Information & Communications Technology Planning & Evaluation (IITP).

REFERENCES

[1] Chris Anderson. 2006. *The long tail: Why the future of business is selling less of more.* Hachette UK.

[2] Akhil Arunkumar, Shin-ying Lee, and Carole-jean Wu. 2016. ID-cache: instruction and memory divergence based cache management for GPUs. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. https://doi.org/10.1109/IISWC.2016.7581276

[3] Rachata Ausavarungnirun, Saugata Ghose, Onur Kayıran, Gabriel H Loh, Chita R Das, Mahmut T Kandemir, and Onur Mutlu. 2018. Holistic management of the GPGPU memory hierarchy to manage warp-level latency tolerance. *arXiv preprint arXiv:1804.11038* (2018).

[4] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

[5] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*. 141–151. https://doi.org/10.1109/IISWC.2012.6402918

[6] Niladrish Chatterjee, Mike O'Connor, Gabriel H. Loh, Nuwan Jayasena, and Rajeev Balasubramonia. 2014. Managing DRAM Latency Divergence in Irregular GPGPU Applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 128–139. https://doi.org/10.1109/SC.2014.16

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[8] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. 2010. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *IEEE International Symposium on Workload Characterization (IISWC'10)*. IEEE, 1–11.

[9] Jack Choquette, Olivier Giroux, and Denis Foley. 2018. Volta: Performance and Programmability. *IEEE Micro* 38, 2 (2018), 42–52. https://doi.org/10.1109/MM.2018.022071134

[10] Saumay Dublish, Vijay Nagarajan, and Nigel Topham. 2017. Evaluating and mitigating bandwidth bottlenecks across the memory hierarchy in GPUs. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 239–248. https://doi.org/10.1109/ISPASS.2017.7975295

[11] Saumay Dublish, Vijay Nagarajan, and Nigel Topham. 2019. Poise: Balancing Thread-Level Parallelism and Memory System Performance in GPUs Using Machine Learning. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 492–505. https://doi.org/10.1109/HPCA.2019.00061

[12] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. 1–10. https://doi.org/10.1109/InPar.2012.6339595

[13] Takakazu Ikeda and Kenji Kise. 2013. Application Aware DRAM Bank Partitioning in CMP. In *2013 International Conference on Parallel and Distributed Systems*. 349–356. https://doi.org/10.1109/ICPADS.2013.56

[14] Shiwei Jia, Ze Tian, Yueyuan Ma, Chenglu Sun, Yimen Zhang, and Yuming Zhang. 2021. A Survey of GPGPU Parallel Processing Architecture Performance Optimization. In *2021 IEEE/ACIS 20th International Fall Conference on Computer and Information Science (ICIS Fall)*. 75–82. https://doi.org/10.1109/ICISFall51598.2021.9627400

[15] Adwait Jog, Onur Kayiran, Ashutosh Pattnaik, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. 2016. Exploiting core criticality for enhanced GPU performance. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*. 351–363.

[16] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*. IEEE, 1–4.

[17] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A Power Modeling Framework for Modern GPUs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 738–753. https://doi.org/10.1145/3466752.3480063

[18] Mahmoud Khairy, Jain Akshay, Tor Aamodt, and Timothy G Rogers. 2018. Exploring modern GPU memory system design challenges through accurate modeling. *arXiv preprint arXiv:1810.07269* (2018).

[19] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[20] Keunsoo Kim, Sangpil Lee, Myung Kuk Yoon, Gunjae Koo, Won Woo Ro, and Murali Annavaram. 2016. Warped-preexecution: A GPU pre-execution approach for improving latency hiding. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 163–175. https://doi.org/10.1109/HPCA.2016.7446062

[21] Gunjae Koo, Hyeran Jeon, and Murali Annavaram. 2015. Revealing Critical Loads and Hidden Data Locality in GPGPU Applications. In *2015 IEEE International Symposium on Workload Characterization*. 120–129. https://doi.org/10.1109/IISWC.2015.23

[22] Gunjae Koo, Hyeran Jeon, Zhenhong Liu, Nam Sung Kim, and Murali Annavaram. 2018. CTA-Aware Prefetching and Scheduling for GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 137–148. https://doi.org/10.1109/IPDPS.2018.00024

[23] Milind Kulkarni, Martin Burtscher, Calin Cascaval, and Keshav Pingali. 2009. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 65–76. https://doi.org/10.1109/ISPASS.2009.4919639

[24] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th annual international symposium on Computer architecture*. 235–246.

[25] Shuai Mu, Yandong Deng, Yubei Chen, Huaiming Li, Jianming Pan, Wenjun Zhang, and Zhihua Wang. 2014. Orchestrating Cache Management and Memory Scheduling for GPGPU Applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 8 (2014), 1803–1814. https://doi.org/10.1109/TVLSI.2013.2278025

[26] Onur Mutlu and Thomas Moscibroda. 2007. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 146–160. https://doi.org/10.1109/MICRO.2007.21

[27] Nvidia. 2015. CUDA C/C++ SDK Code Samples. http://developer.nvidia.com/cuda-cc-sdk-code-samples.

[28] Nvidia. 2019. GeForce RTX 2060 Super. https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2060-super/.

[29] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens. 2000. Memory access scheduling. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*. 128–138. https://doi.org/10.1145/339647.339668

[30] John Sartori and Rakesh Kumar. 2012. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 427–428.

[31] A. Seznec. 1994. Decoupled sectored caches: conciliating low tag implementation cost and low miss ratio. In *Proceedings of 21 International Symposium on Computer Architecture*. 384–393. https://doi.org/10.1109/ISCA.1994.288133

[32] JEDEC standard. 2018. GDDR6. JESD250C.

[33] John A. Stratton, Christopher I. Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Liu, and Wen mei W. Hwu. 2012. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing.

[34] Lu Wang, Magnus Jahre, Almutaz Adileho, and Lieven Eeckhout. 2020. MDM: The GPU Memory Divergence Model. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1009–1021. https://doi.org/10.1109/MICRO50266.2020.00085

[35] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. 2014. Graph processing on GPUs: Where are the bottlenecks?. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 140–149. https://doi.org/10.1109/IISWC.2014.6983053

[36] George L. Yuan, Ali Bakhoda, and Tor M. Aamodt. 2009. Complexity effective memory access scheduling for many-core accelerator architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 34–44. https://doi.org/10.1145/1669112.1669119